L3-INFO Algorithmique et Modélisation

UGA UFR IM²AG Université Grenoble Alpes

Quick – 25 mars 2022 – durée 1 h

Sont interdits : les documents, les ordinateurs, les téléphones (incluant smartphone, tablettes,... tout ce qui contient un dispositif électronique).
Seuls les dictionnaires papier pour les personnes de langue étrangère sont autorisés.
Il sera tenu compte de la qualité de la rédaction et de la clarté de la présentation (2 pts).
En cas d'incompréhension du sujet, préciser les hypothèses de travail que vous faites et continuer.
Le barème **indicatif** : Exercice 1 : 12 pts ($35 \sim mn$) ; Exercice 2 : 6 pts ($20 \sim mn$), Relecture : ($5 \sim mn$)
Les 2 exercices sont indépendants.

I : Randonnée

Que fait un grenoblois lorsqu'il ne fait pas de l'Informatique ? De la montagne évidemment ! Mais il doit faire attention à son paquetage, pas trop lourd mais d'intérêt maximal. Informellement, son problème est de sélectionner des objets parmi un ensemble donné d'objets avec une *valeur* maximale dans le sac sans dépasser la capacité du sac-à-dos.

I.1. Proposer une modélisation de ce problème (Fixer des notations, des paramètres, des structures,...).

> **Éléments de réponse :** On considère un ensemble de n objets, indicés de 1 à n . À chaque objet i on associe une valeur v_i et un poids c_i. On note C la capacité du sac. L'objectif est de trouver un sous-ensemble des objets de valeur maximale tels que les objets rentrent dans le sac, c'est à dire dont le poids total ne dépasse pas C. Si on note $S \subset \{1, \cdots, n\}$ l'ensemble des indices des objets sélectionnés, la valeur du sac est
>
> $$\sum_{i \in S} v_i \text{ et la capacité utilisée } \sum_{i \in S} c_i \leqslant C$$
>
> On aurait pu prendre d'autres conventions/notations.
> Ici on considère que les valeurs, respectivement les poids, sont donnés par un tableau $Valeurs$ respectivement $Poids$. On suppose également que les valeurs et les poids son des nombres (réels) positifs.

Une première approche consiste à mettre les objets dans le sac par "rentabilité" décroissante, la "rentabilité" étant le rapport entre la valeur et le poids.

I.2. Justifier empiriquement cette approche.

> **Éléments de réponse :** Le problème est similaire au rendu de monnaie où on veut rendre un minimum de pièces, rendre la plus grande valeur d'abord semble une bonne heuristique. Ici l'objet qui rapporte le plus est celui de $\frac{v_i}{p_i}$ maximal. Donc mettre cet objet dans le sac et recommencer avec la même stratégie devrait produire une "bonne" solution.
> Ceci revient à construire un algorithme glouton en utilisant un critère d'optimisation local, prendre la rentabilité maximale à chaque étape.

I.3. Montrer que cette approche, dans certains cas, ne fournit pas la meilleure solution.

Éléments de réponse : Comme dans le problème de rendu de monnaie, on cherche un contre-exemple où le premier choix ne permet pas d'obtenir la solution optimale. Prenons $n = 3$ $C = 4$

objet	poids	valeur	rapport v/p
1	3	3	1
2	2	1.8	0.9
3	2	1.8	0.9

Le remplissage proposé va choisir l'objet 1 puis ne pourra rien rajouter, pour une valeur de 3. Si on prend les objets 2 et 3 on aura une valeur de 3.6. Choisir les objets par $\frac{v}{p}$ décroissant ne conduit pas à une solution optimale.

Une solution "bulldozer"

I.4. Écrire un algorithme basé sur l'énumération qui donne toutes les solutions optimales (maximisant la valeur du sac).

Éléments de réponse : On doit ici déterminer un sous-ensemble de l'ensemble des objets. Pour cela on utilise l'algorithme d'énumération de toutes les parties de l'ensemble des objets. Et on adapte les conditions de branchement pour diminuer le coût de calcul.

```
procédure énumérer_visiter_parties (X,S)
  Données : X et S ensembles disjoints d'objets
  Résultat : Une et une seule visite de chaque partie de X ∪ S de la forme p ∪ S avec p ⊂ X et de
             capacité ⩽ C
  if X = ∅
    Visiter (S)
  else
    x =Choisir (X)
    énumérer_visiter_parties (X\{x},S)
    if Condition 2 énumérer_visiter_parties (X\{x},S ∪ {x})
```

— La **Condition 2** permet de garantir que la capacité des objets dans le sac n'excède pas la contrainte. C'est une contrainte de la forme

$$\sum_{i \in S} c_i + c_x \leqslant C$$

Ici l'opération `Visiter` construit une liste des solutions possibles et ne conserve que les solutions de valeur maximale.

On peut également faire l'énumération en gardant en mémoire la meilleure valeur trouvée et couper les branchements qui seront infructueux

```
procédure énumérer_visiter_parties (X,S)
  Données : X et S ensembles disjoints d'éléments
  Résultat : Une et une seule visite de chaque partie de X ∪ S de la forme p ∪ S avec p ⊂ X
  if X = ∅
    Visiter (S)
    Actualiser(val_courante)
  else
    x =Choisir (X)
    if Condition 1 énumérer_visiter_parties (X\{x},S)
    if Condition 2 énumérer_visiter_parties (X\{x},S ∪ {x})
```

UGA UFR IM²AG Université Grenoble Alpes

— La **Condition 1** permet de sélectionner les visite améliorant la valeur de solutions déjà calculées. C'est une contrainte de la forme

$$\sum_{i\in S} v_i + \sum_{i\in X\setminus\{x\}} v_i \geqslant val_courante$$

avec $val_courante$ initialisée à 0. Dans une implémentation plus précise, $val_courante$ et la solution correspondante serait passée en paramètre de la fonction. (le faire en exercice)

— La **Condition 2** permet de garantir que la capacité des objets dans le sac n'excède pas la contrainte. C'est une contrainte de la forme

$$\sum_{i\in S} c_i + c_x \leqslant C$$

Reprendre cet algorithme en effectuant la récursion par passage de référence, et une valeur de retour contenant la solution optimale.

I.5. Donner l'ordre de grandeur de sa complexité.

Éléments de réponse : L'algorithme d'énumération des parties est en 2^n (nombre de sous-ensembles d'un ensemble de n objets). Les conditions permettent de réduire le coût, et dans beaucoup de cas de réduire la complexité, mais au pire on reste en $\mathcal{O}(2^n)$.

Une solution plus fine inspirée du principe de *programmation dynamique*

I.6. En notant $V(k,c)$ la valeur optimale du sac de capacité c obtenue avec les k premiers objets, écrire une équation de récurrence vérifiée par les $V(k,c)$.

Éléments de réponse : Il faut décider si l'objet k est choisi ou non. On a donc

$$V(k,c) = \max\{V(k-1,c), V(k-1,c-c_k)+v_k\}$$

Ici $V(k-1,c)$ est la valeur du sac si l'objet k n'est pas choisi (on a la même capacité résiduelle à remplir c avec les $k-1$ objets restants). Si on choisit l'objet k alors la capacité restante dans le sac est $c-c_k$ et la valeur du sac est augmentée de v_k.
Il faut ainsi calculer $V(n,C)$

I.7. Écrire un algorithme résolvant le problème.

Éléments de réponse :
Sans faire d'optimisation, on construit un tableau de dimension $n\times(C+1)$.
On aura donc 2 itérations imbriquées, l'itération externe considère les objets les uns après lea autres, l'itération interne calcule la valeur du sac si l'objet est choisi pour toutes les capacité c possibles. On distingue le cas où c est plus petite que le poids de l'objet (l'objet ne rentre pas) et la valeur du sac est inchangée.

```
def sac_a_dos_prg_dyn(poids, valeurs, capacite):
    # initialisation du tableau
    resultat = [[0 for j in range(capacite + 1)] for i in range(len(poids))]
    # initialisation de la premiere ligne
    for j in range(poids[0], capacite + 1):
        resultat[0][j] = valeurs[0]
    # calcul des lignes successives
    for i in range(1, len(poids)):
```

```
        # l'objet ne peut pas entrer dans le sac
        for j in range(poids[i]):
            resultat[i][j] = resultat[i - 1][j]
        # l'objet pourrait entrer dans le sac, choisir la meilleure option
        for j in range(poids[i], capacite + 1):
            if resultat[i-1][j] > (resultat[i-1][j - poids[i]] + valeurs[i]):
                resultat[i][j] = resultat[i-1][j]
            else:
                resultat[i][j] = resultat[i-1][j - poids[i]] + valeurs[i]
    return resultat
```

On peut également avoir le même schéma en repliant la mémoire sur un seul vecteur, attention le parcours sur la ligne se fait par indice décroissant

```
def sac_a_dos_prog_dyn_mem(poids, valeurs, capacite):
    # initialisation
    resultat = [ 0 if i < poids[0] else valeurs[0]
                 for i in range(0,capacite + 1)]
    # iteration sur les objets
    for i in range(1,len(poids)):
        for j in range(capacite,poids[i]-1,-1):
            resultat[j] = max(resultat[j],
                              resultat[j-poids[i]]+valeurs[i])
    return resultat
```

On considère l'instance suivante de 6 objets pour un sac de capacité 11 :

objet	poids	valeur
1	2	5
2	3	7
3	5	16
4	4	13
5	7	19
6	3	10

I.8. Expliquer votre algorithme en l'exécutant sur cet exemple.

Éléments de réponse :

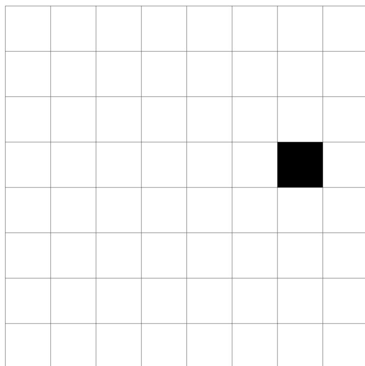
La capacité du sac varie de 0 pour permettre l'initialisation à C

poids	valeur	0	1	2	3	4	5	6	7	8	9	10	11
2	5	0	0	5	5	5	5	5	5	5	5	5	5
3	7	0	0	5	7	7	12	12	12	12	12	12	12
5	16	0	0	5	7	7	16	16	21	23	23	28	28
4	13	0	0	5	7	13	16	18	21	23	29	29	34
7	19	0	0	5	7	13	16	18	21	23	29	29	34
3	10	0	0	5	10	13	16	18	23	26	29	31	34

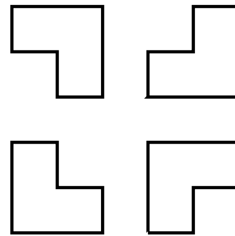
La ligne suivante est le maximum de la ligne précédente avec la ligne précédente décalée du poids de l'objet correspondant à la ligne à laquelle on ajoute la valeur de l'objet.

II : Pavage

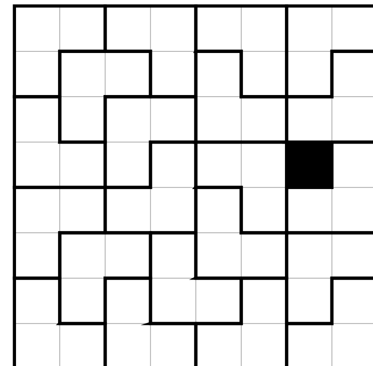
L'objectif est de réaliser le pavage d'un carré troué avec des pièces identiques en forme de L. Pour simplifier l'analyse, on suppose que le carré est de dimension $2^k \times 2^k$, le trou est dans une position arbitraire.



(a) Un exemple de carré à paver



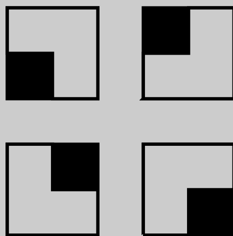
(b) Formes



(c) Un exemple de pavage

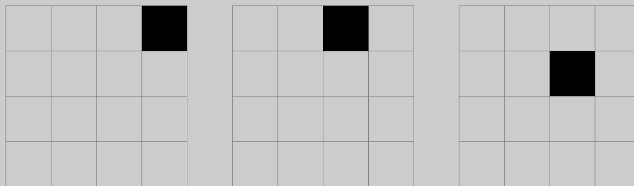
II.1. Résoudre le problème de pavage pour $k = 1$ (carré 2×2).

Éléments de réponse :
Il y a 4 positions possibles du trou dans un carré 2×2

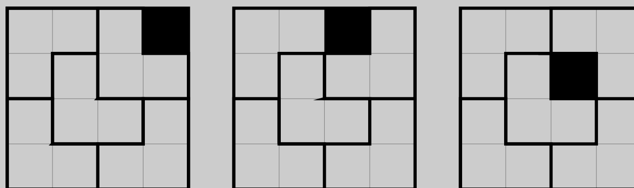


II.2. Résoudre le problème pour $k = 2$ (carré 4×4)

Éléments de réponse :
Il y a 3 positions possibles du trou, aux rotations et symétries près.

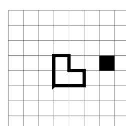


Par essai-erreur on obtient une unique solution pour chaque situation (ce n'est pas vérifié pour toute taille).



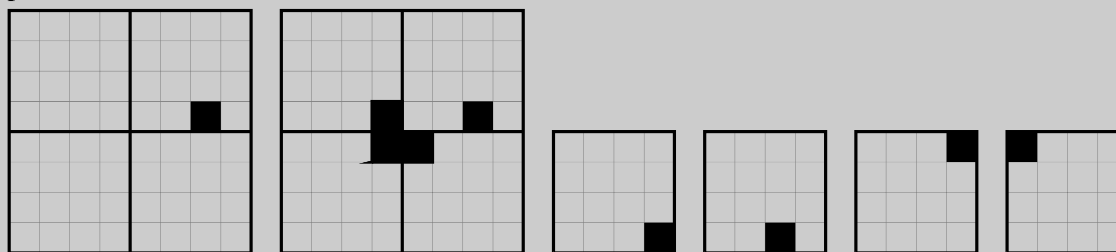
II.3. Écrire un algorithme (récursif), reposant sur le principe du *diviser pour régner*, résolvant le problème de pavage d'un carré troué $2^k \times 2^k$.

Indication : Commencer par placer un L au centre du carré



Éléments de réponse :
Considérons un carré $2^k \times 2^k$, on divise ce carré en 4 carrés de dimension $2^{k-1} \times 2^{k-1}$. Le trou se trouve à l'intérieur de l'un de ces carrés. Les 3 autres carrés forment un grand L.

En plaçant un petit L au creux du grand L on enlève un carré à chacun des 3 carrés de dimension. On se retrouve ainsi à résoudre le même problème de pavage de carré avec un trou mais avec une taille divisée par 4.



II.4. Donner le pavage obtenu après l'exécution de l'algorithme sur l'exemple ci-dessus. On indiquera sur chaque pièce l'ordre dans lequel elle a été posée.

Éléments de réponse :

3			4	8			9
	2				7		
		6					
5				10			11
13			1				19
		14			18		
	12					17	
15			16	20			21

II.5. Calculer la complexité de cet algorithme.

Éléments de réponse : On note $N = 2^k \times 2^k$, on a une équation de complexité

$$T(N) = 4T(\frac{N}{4}) + 1$$

Ce qui nous donne une complexité en $\mathcal{O}(N)$

II.6. Cette méthode peut-elle s'appliquer à un carré quelconque de dimension $n \times n$?

Éléments de réponse : Une condition nécessaire est que $n^2 - 1$ soit divisible par 3. Ce qui est le cas si $n = 2^k$. Au travail pour résoudre, si c'est possible, le carré troué 5×5, puis 7×7. Que dire du carré 9×9 ?